# IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: Kawahito et al
Serial No.:
Filed: herewith

Docket No. JP919990309US1
Art Unit: n/a
Dated:

For: PROGRAM COMPILATION AND OPTIMIZATION

Assistant Commissioner for Patents
Washington, D.C. 20231

## CLAIM OF PRIORITY

Sir:

Applicants, in the above-identified application, hereby claim the right of priority in connection with Title 35 U.S.C. &119 and in support thereof, herewith submits a certified copy of Japanese Patent Application No. 2000-114193, filed on April 14, 2000.

Respectfully submitted,

Manny W. Schecter
Reg. No. 31,722
Tel. (914) 945-3252

IBM CORPORATION
INTELLECTUAL PROPERTY LAW DEPT.
P.O. BOX 218
YORKTOWN HEIGHTS, NEW YORK 10598

ExpressMail EL715831397US
Date of Deposit:March 21,2001

JP9199900309

# 日 本 国 特 許 庁
PATENT OFFICE
JAPANESE GOVERNMENT

別紙添付の書類に記載されている事項は下記の出願書類に記載されてる事項と同一であることを証明する。
This is to certify that the annexed is a true copy of the following application as filed h this Office.



出願年月日
Date of Application: ２０００年　４月１４日

出願番号
Application Number: 特願２０００－１１４１９３

出願人
Applicant (s): インターナショナル・ビジネス・マシーンズ・コーポレーション

CERTIFIED COPY OF
PRIORITY DOCUMENT

２０００年１２月２２日

特許庁長官
Commissioner,
Patent Office 及川耕造

出証番号　出証特２０００－３１０７２５８

【書類名】　特許願

【整理番号】　JA999309

【提出日】　平成12年 4月14日

【あて先】　特許庁長官　殿

【国際特許分類】　G06F　5/06

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　川人　基弘

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　小笠原　武史

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　小松　秀昭

【特許出願人】

【識別番号】　390009531

【氏名又は名称】　インターナショナル・ビジネス・マシーンズ・コーポレイション

【代理人】

【識別番号】　100086243

【弁理士】

【氏名又は名称】　坂口　博

【復代理人】

【識別番号】　100104880

【弁理士】

【氏名又は名称】　古部　次郎

【選任した代理人】

【識別番号】　100091568

【弁理士】

【氏名又は名称】　市位　嘉宏

【選任した復代理人】

【識別番号】　100100077

【弁理士】

【氏名又は名称】　大場　充

【手数料の表示】

【予納台帳番号】　081504

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【包括委任状番号】　9706050

【包括委任状番号】　9704733

【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　コンパイラ、コンピュータシステム、最適化方法、コンピュータプログラム、記憶媒体及びプログラム伝送装置

【特許請求の範囲】

【請求項１】　プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、

機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、

前記オブジェクトプログラムに対して、当該オブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点と当該例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴とするコンパイラ。

【請求項２】　前記プログラム変形部は、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令に関して、前記例外処理の発生点と前記例外処理を実行する場所との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを生成し、前記オブジェクトプログラムに挿入することを特徴とする請求項１に記載のコンパイラ。

【請求項３】　前記プログラム変形部は、

前記最適化処理実行部による最適化処理に先立って、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定する前処理部と、

前記最適化処理実行部により最適化処理を施された前記オブジェクトプログラムにおける前記例外処理が起きる可能性のある命令に対して、当該例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、当該内容的な相違を補償する補償コードと、当該補償コードを経た後に当該例外処理を実行する場所に制御を移すためのコードとを、前記キャッチ節の中に生成する後処理部とからなることを特徴とする請求項１に記載のコンパイラ。

【請求項４】　前記プログラム変形部は、

前記最適化処理実行部による最適化処理に先立って、前記オブジェクトプログラム中の例外処理が起きる可能性のある命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、当該例外処理が起きた場合は当該実際に例外処理を起こす命令に制御を移すように、前記オブジェクトプログラムを変形することを特徴とする請求項１に記載のコンパイラ。

【請求項５】　前記プログラム変形部は、

前記オブジェクトプログラム中の例外処理が起きる可能性のある命令が、例外処理が起きるかどうかをチェックする部分と実際に例外処理を起こす部分とで構成されている場合は、前記最適化処理実行部による最適化処理に先立って、当該命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、当該例外処理が起きた場合は当該実際に例外処理を起こす命令に制御を移すように、前記オブジェクトプログラムを変形し、

その他の場合は、前記最適化処理実行部による最適化処理に先立って、前記例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定し、

前記最適化処理実行部により最適化処理を施された前記オブジェクトプログラムにおける前記例外処理が起きる可能性のある命令に対して、当該例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、当該内容的な相違を補償する補償コードと、当該補償コードを経た後に当該例外処理を実行する場所に制御を移すためのコードとを、前記キャッチ節の中に生成することを特徴とする請求項１に記載のコンパイラ。

【請求項６】　プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラを備えたコンピュータシステムにおいて、

前記コンパイラは、

機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、

前記オブジェクトプログラムに対して、当該オブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点と当該例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴とするコンピュータシステム。

【請求項７】　プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成し、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成することを特徴とする最適化方法。

【請求項８】　プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを含むベーシックブロックを設定するステップと、

前記ベーシックブロックが設定された前記プログラムに対して最適化処理を実行するステップと、

前記最適化処理の施された前記プログラムにおいて、例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間において、内容における相違があるか否かをチェックするステップと、

前記内容における相違がある場合に、前記ベーシックブロックの前記キャッチ節内に、当該相違を補償する補償コードと、補償コードを経た後に前記例外処理を実行する場所に制御を移すためのコードとを生成するステップとを含むことを特徴とする最適化方法。

【請求項９】　前記例外処理が起きる可能性のある命令と当該例外処理を実

行する場所との間における内容的な相違をチェックするステップにおいて、内容が相違しない場合に、当該例外処理が起きる可能性のある命令に対する前記ベーシックブロックを除去するステップをさらに含むことを特徴とする請求項８に記載の最適化方法。

【請求項１０】　プログラムの処理効率を向上させるための最適化を行う最適化方法において、

処理対象であるプログラム中の例外処理が起きる可能性のあるコードを、例外処理が起きるかどうかをチェックするコードと、実際に例外処理を起こすコードとに分割するステップと、

前記コードを分割するステップで分割されたコードをコントロールフローグラフの分岐として明示するステップと、

例外処理が起きた場合に、前記実際に例外処理を起こすコードに制御を移すように前記コントロールフローグラフを構成するステップと、

変形の行われた前記プログラムに対して最適化処理を実行するステップとを含むことを特徴とする最適化方法。

【請求項１１】　前記最適化処理を実行した後、前記実際に例外処理を起こすコードを含むブロックに、例外処理の発生点と前記実際に例外処理を起こすコードとの間における内容的な相違を補償するコードが生成されているかどうかをチェックするステップと、

前記内容的な相違を補償するコードが生成されていない場合に、前記分割されたコード及びコントロールフローグラフを合成して分割前の状態に戻すステップとをさらに含むことを特徴とする請求項１０に記載の最適化方法。

【請求項１２】　コンピュータにオブジェクトプログラムの最適化処理を実行させるコンピュータプログラムにおいて、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違が

ある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを前記コンピュータに実行させることを特徴とするコンピュータプログラム。

【請求項１３】　コンピュータに実行させるコンピュータプログラムにおいて、

実行時に、前記コンピュータプログラム中の例外処理が起きる可能性のある命令に対して、

前記例外処理が起きるかどうかをチェックする機能と、

前記例外処理を実行する機能と、

前記例外処理が起きた場合に前記例外処理を実行する部分に制御を移すと共に、前記例外処理の発生点と前記例外処理を実行する部分との間に内容における相違がある場合に、前記例外処理を実行する部分への移行に先立って当該内容における相違を補償する機能とを備えることを特徴とするコンピュータプログラム。

【請求項１４】　前記例外処理が起きるかどうかチェックする機能は、トライーキャッチブロックのトライ節または条件分岐であることを特徴とする請求項１３に記載のコンピュータプログラム。

【請求項１５】　コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、

前記プログラムは、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項１６】　コンピュータに、

処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、当該

例外処理が起きるかどうかをチェックする部分を含み、かつ当該例外処理が起きた場合に当該例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、

前記例外処理の発生点と前記例外処理を実行する部分との間に内容的な相違がある場合に、当該内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成する処理とを実行させるプログラムを記憶する記憶手段と、

前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化処理を行うことが可能なコンパイラ及び最適化方法に関する。

【０００２】

【従来の技術】

通常、プログラミング言語で記述されたプログラムのソース・コードをコンパイルする際、ソースプログラムから変換された機械語のオブジェクトプログラムに対して、実際の処理効率を向上させるための最適化処理を行っている。

【０００３】

ところで、今日のプログラミング言語の多くは、仕様において、ユーザーが定義したエクセプション・ハンドラがある場合には、例外処理（エクセプション）が起きたときに、そのエクセプション・ハンドラに処理を移すことを規定している。また、例外処理が起きる可能性のある命令の数も多くなっている。

【０００４】

このようなプログラミング言語においては、例外処理が起きた場合、それまでの実行状況を保証する必要がある。すなわち、例外処理が起きる可能性自体は低いが、これを無視することはできないため、例外処理が起きた場合に対応するための処理を用意しておかなくてはならない。しかし、例外処理が起きる可能性が低いということは、そのための対応処理が必要となることも少ないため、全体と

して動作効率が落ちることとなる。そのため、プログラム中に例外処理が起きる可能性のある命令がある場合とそのような命令が無い場合とを比べると、上述したプログラムのコンパイル時における最適化の効果が低下していた。

【０００５】

また従来、エクセプション・ハンドラを含む関数については、トライ領域（Try Region）内で通常のレジスタ割付を行うと、例外処理が起きた場合に、変数の値がどこに格納されているかを特定することができなかった。このため、エクセプション・ハンドラ以降で参照がある変数については、例外処理が起きたか否かに関わらず、基本的にメモリ上にあるものとして扱っていた。したがって、例外処理が起きない場合でも、当該変数をメモリ上に読みに行かなければならないため、全体的な実行速度を低下させることとなっていた。

【０００６】

このようなプログラムの最適化に対する弊害を防止するため、プログラムにおいて例外処理が起きる可能性のある命令を減らすような技術が、従来から提案されている。そのような従来技術としては、例えば、ループのバージョニングやデータフローアナリシスを用いたエクセプション・チェックの除去等がある。

これらの従来技術によれば、例外処理が起きる可能性がある中間コードそのものがなくなるため、エクセプション・チェックのコストを削減することができ、それだけでなく各種の最適化処理の効果を高めることができる。

【０００７】

【発明が解決しようとする課題】

上述したように、例外処理に対してエクセプション・ハンドラを設定して処理を移行するプログラミング言語にて記述されたプログラムは、プログラム中に例外処理が起きる可能性のある命令を含む場合、最適化の効率が悪かった。

【０００８】

これに対し、従来から提案されているプログラムにおいて例外処理が起きるがある命令を減らす手法により、例外処理が起きる可能性のある中間コードを減らし、最適化の効果を高めることが可能である。

しかし、この従来の手法は、全ての例外処理が起きる可能性のある中間コード

を除去できるわけではなく、例えば、関数呼び出しの命令等は除去できなかった。そのため、除去できない命令に関しては、依然として最適化の効果を低下させる原因となっていた。

【０００９】

そこで本発明は、例外処理が起きる可能性のある命令を含むプログラムに対して効果の高い最適化を行うことを目的とする。

【００１０】

【課題を解決するための手段】

かかる目的のもと、本発明は、プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴としている。

これにより、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常の最適化処理を実行することが可能となる。

【００１１】

ここで、このプログラム変形部は、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関して、この例外処理の発生点とこの例外処理を実行する場所との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードを生成し、このオブジェクトプログラムに挿入することを特徴としている。

例外処理の発生点とこの例外処理を実行する場所との間における内容的な相違とは、例えばレジスタ割付やメモリ内容的な相違である。これらの具体的な相違の内容に応じて、適切な補償コードを挿入することとなる。

【００１２】

またここで、このプログラム変形部は、最適化処理実行部による最適化処理に

先立って、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定する前処理部と、最適化処理実行部により最適化処理を施されたオブジェクトプログラムにおける例外処理が起きる可能性のある命令に対して、この例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、この内容的な相違を補償する補償コードと、この補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを、キャッチ節の中に生成する後処理部とからなることを特徴としている。

例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違がない場合には、前処理部で作成したトライ節とキャッチ節とを削除することができる。

【００１３】

さらにここで、このプログラム変形部は、前記最適化処理実行部による最適化処理に先立って、このオブジェクトプログラム中の例外処理が起きる可能性のある命令を例外処理が起きるかどうかをチェックする命令と実際に例外処理を起こす命令とに分割すると共に、この例外処理が起きた場合はこの実際に例外処理を起こす命令に制御を移すように、このオブジェクトプログラムを変形することを特徴としている。

このように変形したオブジェクトプログラムに対して最適化処理を行った後に、実際に例外処理を起こす命令を含むブロックにおいて、例外処理の発生点とこの例外処理を実行する場所との間における内容的な相違を補償するためのコードが生成されていなければ、分割した命令を合成し、分割前の命令に戻すことができる。

【００１４】

さらにここで、このプログラム変形部は、このオブジェクトプログラム中の例外処理が起きる可能性のある命令が、例外処理が起きるかどうかをチェックする部分と実際に例外処理を起こす部分とで構成されている場合は、最適化処理実行部による最適化処理に先立って、この命令を例外処理が起きるかどうかをチェッ

クする命令と実際に例外処理を起こす命令とに分割すると共に、この例外処理が起きた場合はこの実際に例外処理を起こす命令に制御を移すように、このオブジェクトプログラムを変形し、その他の場合は、最適化処理実行部による最適化処理に先立って、この例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを設定し、かつ最適化処理実行部により最適化処理を施されたこのオブジェクトプログラムにおけるこの例外処理が起きる可能性のある命令に対して、この例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間に内容的な相違があるか否かを調べ、相違がある場合に、この内容的な相違を補償する補償コードと、この補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを、キャッチ節の中に生成することを特徴としている。

【００１５】

また、本発明は、プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラを備えたコンピュータシステムにおいて、このコンパイラは、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行うプログラム変形部とを備えることを特徴としている。

このような構成とすれば、このコンピュータシステムにおいて、プログラムをコンパイルする場合に、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常の最適化処理を実行することが可能となる点で好ましい。

【００１６】

さらにまた、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させ

る命令を持つベーシックブロックを作成し、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードを前記ベーシックブロックの中に生成することを特徴としている。

これにより、例外処理が起きる可能性のある命令において、例外処理が起きた場合を考慮して特別に扱うこと無く、通常の最適化処理を実行することが可能となる。

【００１７】

さらに、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対して、例外処理が起きるかどうかをチェックするトライ節と、例外処理が発生した場合に固有の処理を行うためのキャッチ節とを含むベーシックブロックを設定するステップと、このベーシックブロックが設定されたプログラムに対して最適化処理を実行するステップと、この最適化処理の施されたプログラムにおいて、例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間において、内容における相違があるか否かをチェックするステップと、この内容における相違がある場合に、このベーシックブロックのキャッチ節内に、この相違を補償する補償コードと、補償コードを経た後にこの例外処理を実行する場所に制御を移すためのコードとを生成するステップとを含むことを特徴としている。

最適化処理の前にベーシックブロックを設定しておき、最適化の後に、補償の必要性をチェックして補償コードを生成することにより、例外処理が起きる可能性のある命令とこの例外処理を実行する場所との間における内容的な相違を適切に補償することができる。

【００１８】

ここで、さらに、この例外処理が起きる可能性のある命令と当該例外処理を実行する場所との間における内容的な相違をチェックするステップにおいて、内容が相違しない場合に、当該例外処理が起きる可能性のある命令に対する前記ベーシックブロックを除去するステップを含むこととすることができる。

不要なベーシックブロックを除去することにより、最適化後のオブジェクトプログラムのコード量を削減することができる。

【００１９】

また、本発明は、プログラムの処理効率を向上させるための最適化を行う最適化方法において、処理対象であるプログラム中の例外処理が起きる可能性のあるコードを、例外処理が起きるかどうかをチェックするコードと、実際に例外処理を起こすコードとに分割するステップと、コードを分割するステップで分割されたコードをコントロールフローグラフの分岐として明示するステップと、例外処理が起きた場合に、この実際に例外処理を起こすコードに制御を移すようにこのコントロールフローグラフを構成するステップと、この変形の行われたプログラムに対して最適化処理を実行するステップとを含むことを特徴としている。

このように変形されたプログラムは、例外処理を起こすコードにおいて、必要に応じて、例外処理の発生点と例外処理を実行する場所との間における内容的な相違を補償するコードが含まれている。したがって、この変形の後にこのプログラムに対して最適化処理を行うことにより、効率的な最適化が実現される。

【００２０】

ここで、さらに、最適化処理を実行した後、実際に例外処理を起こすコードを含むブロックに、例外処理の発生点とこの実際に例外処理を起こすコードとの間における内容的な相違を補償するコードが生成されているかどうかをチェックするステップと、この内容的な相違を補償するコードが生成されていない場合に、この分割されたコード及びコントロールフローグラフを合成して分割前の状態に戻すステップとを含むこととすることができる。

無意味な分岐を元の状態に戻すことにより、最適化後のオブジェクトプログラムのコード量を削減することができる。

【００２１】

また、本発明は、コンピュータにオブジェクトプログラムの最適化処理を実行させるコンピュータプログラムにおいて、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分

に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベーシックブロックの中に生成する処理とをコンピュータに実行させることを特徴としている。

このような構成とすることにより、このプログラムをインストールしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【００２２】

また、本発明は、コンピュータに実行させるコンピュータプログラムにおいて、実行時に、このコンピュータプログラム中の例外処理が起きる可能性のある命令に対して、この例外処理が起きるかどうかをチェックする機能と、この例外処理を実行する機能と、この例外処理が起きた場合にこの例外処理を実行する部分に制御を移すと共に、この例外処理の発生点とこの例外処理を実行する部分との間に内容における相違がある場合に、この例外処理を実行する部分への移行に先立ってこの内容における相違を補償する機能とを備えることを特徴としている。

このように構成されたプログラムは、例外処理が起きる可能性のある命令を含んでいても、このような命令を、例外処理を考慮して特別に扱うことなく最適化できるため、実行速度が向上する点で優れている。

【００２３】

ここで、例外処理が起きるかどうかチェックする機能は、トライーキャッチブロックのトライ節または条件分岐により実現することができる。

【００２４】

また、本発明は、コンピュータに実行させるプログラムをこのコンピュータの入力手段が読取可能に記憶した記憶媒体において、このプログラムは、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベ

ーシックブロックの中に生成する処理とをこのコンピュータに実行させることを特徴としている。

このような構成とすることにより、このプログラムをインストールしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【００２５】

コンピュータに、処理対象であるプログラム中の例外処理が起きる可能性のある命令に対し、この例外処理が起きるかどうかをチェックする部分を含み、かつこの例外処理が起きた場合にこの例外処理を実行する部分に制御を移行させる命令を持つベーシックブロックを作成する処理と、この例外処理の発生点とこの例外処理を実行する部分との間に内容的な相違がある場合に、この内容的な相違を補償する補償コードをこのベーシックブロックの中に生成する処理とを実行させるプログラムを記憶する記憶手段と、この記憶手段からこのプログラムを読み出してこのプログラムを送信する送信手段とを備えたことを特徴としている。

このような構成とすることにより、このプログラムをダウンロードしたコンピュータにおいて、例外処理が起きる可能性のある命令を含むプログラムに対して効果的な最適化を行うことが可能となる。

【００２６】

【発明の実施の形態】

以下、添付図面に示す実施の形態に基づいてこの発明を詳細に説明する。

まず、本発明の概要を説明する。なお、以下では、オブジェクト指向プログラミング言語で記述されたプログラムを前提として説明する。しかし、本発明による最適化技術の適用範囲は、ユーザーが定義したエクセプション・ハンドラがある場合に、例外処理が起きたならばそのエクセプション・ハンドラに処理を移す仕様を持ったプログラミング全般に及び、オブジェクト指向プログラミング言語に限定されるものではない。

本発明の最適化方法は、例外処理が起きる可能性のある命令を含むプログラムにおいて、例外発生点とエクセプション・ハンドラ（メソッド外も含む）までの間に、最適化による２点の違い（例えばレジスタ割付やメモリ内容）を吸収する

ための補償コードを生成する。この変形により、当該プログラムの実行時において、例外発生時には、システムは例外発生点から補償コードの存在を確認し、存在するならば、当該エクセプション・ハンドラの代わりに、まず補償コード点へ飛び、その後エクセプション・ハンドラに制御を移す。

【００２７】

図１は、例外処理が起きる可能性のある命令において、エクセプション・ハンドラへ移行する様子を示す図である。図１において、例外処理１では、例外処理が発生しても補償コードを要することなくエクセプション・ハンドラへ移行する。これに対し、例外処理２、３では、例外発生点とエクセプション・ハンドラとの間の違いを吸収するために、補償コードを経てエクセプション・ハンドラへ移行する。このように、必要に応じて例外発生点から直接エクセプション・ハンドラへ移行するのではなく、補償コードを介することにより、例外処理１、２、３のいずれにおいても、エクセプション・ハンドラにおいては同じレジスタイメージを保証することができる。

【００２８】

これにより、例外処理が起きる可能性のある命令において、例外処理が発生した場合のことを考慮して、変数をメモリに読みに行くなどの特別な扱いを行う必要が無くなる。したがって、例外処理が起きる可能性のある命令を特別に扱わず最適化できるため、例外処理が発生しない場合の最適化の効率を向上させることができ、全体的な実行速度を向上させることができる。

【００２９】

補償コードの挿入は、プログラム中にベーシックブロックを作成し、当該ベーシックブロック中に当該補償コードを生成することにより行われる。このベーシックブロックによる補償コードを挿入する場所を作成する方法として、Try-Catchブロックを作成する方法と、明示的に条件分岐を設定する方法とがある。以下、各方法を分けて説明する。

【００３０】

〔第１の実施の形態〕

図２は、第１の実施の形態によるコンパイラにおける最適化処理部の構成を説

明する図である。

図２において、符号１０は最適化処理部であり、機械語のオブジェクトコード（中間コード）に変換されたプログラムを最適化する。符号１１は前処理部であり、補償コードを生成するためのベーシックブロックを作成する。符号１２は最適化処理実行部であり、前処理部１１により前処理の施されたオブジェクトプログラムに対して通常の最適化処理を実行する。符号１３は後処理部であり、前処理部１１にて作成されたベーシックブロックの中に例外処理に対応するための補償コードを設定する。

【００３１】

本実施の形態において、前処理部１１は、プログラム中の例外処理が起きる可能性のある命令に対して、ベーシックブロックとしてTry-Catchブロックを設定する。ここで、Try-Catchブロックは、例外処理が起きる可能性のある命令をＴｒｙ節とＣａｔｃｈ節とで挟む。そして、Ｔｒｙ節で例外処理が起きたか否かをチェックし、例外処理が発生したならば、Ｃａｔｃｈ節の処理に移行する。したがって、このＣａｔｃｈ節の中に補償コードが生成されることとなる。

【００３２】

最適化処理実行部１２は、通常の最適化処理を実行する。最適化処理の対象であるプログラムは、前処理部１１により例外処理が起きる可能性のある命令に対して、Try-Catchブロックを設けているため、全体としては例外処理を考慮して特別に扱う必要がない。したがって、効果の高い最適化を実現することができる。

【００３３】

後処理部１３は、Try-CatchブロックのＣａｔｃｈ節に必要な補償コードを生成する。また、補償コードを必要としない場合は、前処理部１１にて作成されたTry-Catchブロックを除去する。ここで、補償コードを必要としない場合とは、例外処理が起きる可能性のある命令と、本来の当該例外処理を実行する場所であるエクセプション・ハンドラとの間に相違点がない場合である。この場合、図１に示した例外処理１のように、そのままエクセプション・ハンドラへ移行すればよい。したがって、補償コードは必要なく、本実施の形態によるTry-Catchブロ

ックも不要となるため、プログラムから除去する。

【００３４】

図３は、本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

図３を参照すると、オブジェクトコード（中間コード）に変換されたプログラムを入力した最適化処理部１０において、まず、前処理部１１が、メソッド内の例外処理が起きる可能性のある命令について補償コード生成のためにTry-Catchブロックを設定する（ステップ３０１）。

次に、最適化処理実行部１２が、最適化処理を実行する（ステップ３０２）。

次に、後処理部１３が、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間において、レジスタイメージやメモリ内容等に相違点があるか否かをチェックする（ステップ３０３）。そして、相違点があるならば、前処理部１１において作成したTry-CatchブロックのＣａｔｃｈ節内に当該相違を補償する補償コードと、補償コードを経た後にエクセプション・ハンドラに制御を移すためのコードとを生成する（ステップ３０４、３０５）。

また、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間に相違点がないならば、当該Try-Catchブロックを除去する（ステップ３０３、３０６）。

【００３５】

本実施の形態によれば、例外処理が起きる可能性のある命令について、プログラムの実行時に例外処理が起きない場合は、本実施の形態を適用しない場合と比べて同等以上の実行速度が得られる。これに対し、実行時に例外処理が起きた場合は、本実施の形態を適用しない場合と比べて遅くなる。しかしながら、例外処理が起きる可能性のある命令について、例外処理が起きない頻度は、例外処理が起きる頻度をかなり上回ると予想され、無視することができる。

【００３６】

ここで、例外処理が発生した場合に補償コードを見つける手法について説明する。上述したように、本実施の形態では、補償コードを挿入するためにTry-Catchブロックを用いる。しかし、従来のTry-Catchの実現方法として、Ｔｒｙ節一つ

に対してエクセプション・ハンドラを対応させる方法があった。この方法では、Ｔｒｙ節の境界でＴｒｙ情報を記憶しておき、例外処理の発生時には、記憶したＴｒｙ情報に基づいてエクセプション・ハンドラを検出していた。

しかし、本実施の形態を適用すると、メソッド内の複数のエクセプションが起きる可能性がある命令に対して、補償コードを挿入するためのTry-Catchブロックが生成される。この時、従来のTry-Catchの実現方法を用いて補償コードを見つけるようにすると、Ｔｒｙ節の境界でＴｒｙ情報を記憶するために実行速度が遅くなってしまう。

そこで、本実施の形態では、Try-Catchの実現方法として例外発生点のアドレスと例外処理の種類に基づいて、対応するエクセプション・ハンドラを検出し、制御を移すような機構を導入する。この機構を導入することにより、実行速度が遅くなることを回避することができる。

【００３７】

次に、Try-Catchブロックにおいて例外発生点からエクセプション・ハンドラに制御を移すための手法について説明する。

まず、コンパイル時に、後処理部１３が、トライ領域内の例外処理が起きる可能性のある命令について例外発生点のアドレス、対応するハンドラのアドレス、ハンドラのエクセプションの種類を登録する。その命令で起こった例外処理により制御が移る可能性があるエクセプション・ハンドラが複数ある時は、対応するハンドラごとにこれらの情報を全て登録する。

【００３８】

次に、プログラムの実行時に、以下の手順で、例外発生コードからエクセプション・ハンドラに制御を移すための処理を行う。

まず、メソッド内で例外処理が起きたときには、登録されている例外発生点のアドレスと例外処理の種類とを検索し、対応するエクセプション・ハンドラのアドレスを取得する。

そして、検索に成功したならば、検出したエクセプション・ハンドラに制御を移す。

一方、検索に失敗したならば、現メソッドを呼んだメソッドに制御を移し、検

索を続行する。

この方法を適用した場合、例外処理が起きない場合は、従来のトライ領域の境界で記録する方法と比べて高速に実行することが可能である。一方、登録テーブルを生成するため、従来と比べて、コンパイラが生成するコード量が増えるおそれがある。しかしながら、上述したように、後処理部１３において、例外処理が起きる可能性がある命令とエクセプション・ハンドラとの間に相違点がない場合、すなわち最適化の効果が無かった変形は、元の形に戻すようにしている。このため、登録テーブルのサイズも最小限必要なものに限られている。したがって、コード量の増大も最小限に抑えることができる。

【００３９】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する具体的な実施例を説明する。

まず、エクセプション・ハンドラ以降で参照がある変数に関して、トライ領域内で通常のレジスタ割付を使って、当該変数をレジスタに割り付けることを可能にする方法について説明する。

【００４０】

本実施の形態による補償コードの挿入を、次のように適用することによって、エクセプション・ハンドラ以降で参照がある変数に関して、トライ領域内で通常のレジスタ割付を使って、当該変数をレジスタに割り付けることが可能となる。また、例外処理が起きた場合でも、コンパイラにより生成されたエクセプション・ハンドラ内の補償コードにより、正しく動作させることができる。

まず、エクセプション・ハンドラ以降で参照がある変数の集合情報をトライ領域ごとに求める。また、トライ領域内ではエクセプションが起きる命令についてエクセプション・ハンドラを設定する。

さらに、エクセプション・ハンドラ内において、最初に求めた集合情報に基づいて補正コードである調整コードを生成する。

そして、エクセプション・ハンドラの最後で、本来移行するはずだったエクセプション・ハンドラに制御を移す。

【００４１】

以上の動作を、図４乃至図７の擬似コードを参照しながらさらに詳細に説明する。なお、以下の説明において、変数の値設定をＤＥＦ、変数の使用をＵＳＥ、トライ領域の識別番号をＴＲＹとする。

まず、個々のトライ領域に対応するエクセプション・ハンドラの開始点で、全てのローカル変数についてＤＥＦがあると仮定し、そのＤＥＦを参照するローカル変数の集合：USE_VIA_EH_LIST[TRY] を求める。この動作は、次の三つの動作からなる。なお、以下の説明及び図示のコードにおいて、reach[bbidx].in、reach[bbidx].out、reach[bbidx].gen、reach[bbidx].killは、トライ領域の識別番号およびローカル変数番号の２つを持つ要素からなる集合である。

【００４２】

Ａ．図４に示すアルゴリズムを用い、各ベーシックブロックのreach[bbidx].gen及びreach[bbidx].killを求める。

Ｂ．Ａ．で求めたgen及びkillを用い、図５に示す式でデータフローアナリシスを行い、エクセプション・ハンドラの開始点で設定したＤＥＦが各ベーシックブロックの先頭に到達する集合reach[B].inを求める。

Ｃ．Ｂ．で求めたreach[B].inを用い、図６に示すアルゴリズムにより、エクセプション・ハンドラの開始点で設定したＤＥＦを参照するローカル変数の集合 USE_VIA_EH_LIST[TRY] を求める。

【００４３】

次に、トライ領域内で対応するエクセプション・ハンドラに到達する可能性がある例外処理が起きる命令に対して、図７に示すアルゴリズムを用いて、新たなエクセプション・ハンドラを設定する。

図８は、図７のアルゴリズムにおける

＜この命令について新たなエクセプション・ハンドラを設定する。＞

の処理内容を説明するフローチャートである。

図８を参照すると、まず、USE_VIA_EH_LIST[TRY]内のローカル変数に対して、本来のエクセプション・ハンドラの入り口でのローカル変数のレジスタイメージを決める（ステップ８０１）。そして、当該レジスタイメージに合うように、メモリまたは特定のレジスタへローカル変数値をコピーするコードの集合情報を求

める（ステップ８０２）。

【００４４】

次に、求めた集合情報が空集合かどうかを調べる（ステップ８０３）。空集合でなければ、次に、新しいベーシックブロックを作成する（ステップ８０４）。そして、作成したベーシックブロック内に、ステップ８０２で作成した集合情報のコードを生成する（ステップ８０５）。

次に、当該ベーシックブロックに対して、ステップ８０５で生成したコードの後に、本来のエクセプション・ハンドラに制御を移すためのコードを生成する（ステップ８０６）。

【００４５】

最後に、例外処理を起こす可能性がある命令に対して、Try-Catchブロックを作成し、Ｃａｔｃｈ点として、ステップ８０４で作成したベーシックブロックを設定する（ステップ８０７）。

なお、ステップ８０３において、ステップ８０２で求めた集合情報が空集合であったならば、ステップ８０４以降の動作を行わずに終了する。

以上の処理を行うことにより、エクセプション・ハンドラ以降で参照がある変数についても、トライ領域内で通常のレジスタ割付を使ってレジスタに割り付けることが可能になる。

【００４６】

次に、本実施の形態における補償コードの挿入によりプログラムの実行速度が速くなる具体例を示す。

図９は、配列内の最小値及び最大値を求めるＪａｖａのサンプルプログラムを示す図である。また、図１０は、図９のプログラムに対して本実施の形態により補償コードを挿入する変形を加えた状態を示す図である。

【００４７】

図９に示すプログラムにおいて、まず、エクセプション・ハンドラの開始点で設定したＤＥＦを参照するローカル変数の集合：USE_VIA_EH_LIST[TRY]を求める。

ローカル変数ｉ及びｊは、プログラム中の（１６）行でＵＳＥがある。また、

最小値ｍｉｎ及び最大値ｍａｘは、（１９）行、（２０）行でＵＳＥがあるが（１７）行、（１８）行において、エクセプション・ハンドラの開始点で設定したＤＥＦは壊される。

また、ａ、size_x、size_yに関しては、エクセプション・ハンドラ以降にＵＳＥはない。よって、USE_VIA_EH_LIST[TRY]は、ｉ及びｊとなる。この２つの変数について、エクセプション・ハンドラの入り口でのレジスタイメージは、それぞれＲ１、Ｒ２とする。

【００４８】

次に、新たなエクセプション・ハンドラを設定する。

トライ領域内で、対応するエクセプション・ハンドラに到達する可能性がある例外処理が起きる命令は、（３）行及び（９）行である。これらの命令に対してエクセプション・ハンドラを設定すると、図１０のように変形される。図１０を参照すると、（３）行は（３．１）行から（３．４）行まで、（９）行は（９．１）行から（９．３）行までにそれぞれ示すように、命令を細かい単位に分けてある。なお、NULLCHECKとはNULLかどうかをチェックする中間コードを表し、SIZECHECKとは配列のサイズとインデックスとをチェックする中間コードを表している。

【００４９】

図９のプログラムを図１０に示すように変形することによって、トライ領域内における例外処理が起きないパスにおいて、全ての変数を通常のレジスタ割付を使ってレジスタに割り付けることが可能になる。

【００５０】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する他の実施例として、部分的な冗長度除去（Partial Redundancy Elimination）のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法について説明する。

【００５１】

図１１は、本実施例における動作を説明するフローチャートである。

図１１を参照すると、まず、例外処理が起きる可能性のある中間コードに対し

、新たなエクセプション・ハンドラを設定する（ステップ１１０１）。当該エクセプション・ハンドラは、当該中間コードによって起き得る全てのエクセプションをＣａｔｃｈするように設定する。

【００５２】

例外処理が起きる可能性のある中間コードにおける当該例外処理が１種類であり、かつ対応するエクセプション・ハンドラが元々メソッド内にあるならば、Ｃａｔｃｈ節の中に、補償コードとして、当該エクセプション・ハンドラにジャンプするコードを生成する。それ以外の場合は、受け取った例外処理をThrowするコードを生成する（ステップ１１０２）。

【００５３】

また、上記の例外処理が起きる可能性のある中間コードの下でベーシックブロックを分割し、Ｃａｔｃｈ節のベーシックブロックと例外処理が起きる可能性のある中間コードに後続する中間コードとの二つの間にコントロールフローグラフのエッジを張る（ステップ１１０３）。

【００５４】

次に、ステップ１１０３で作成されたコントロールフローグラフに対して、メモリへの書き込み命令の最適化を行う（ステップ１１０４）。

最後に、ステップ１１０１で作成したエクセプション・ハンドラ に対して、Ｃａｔｃｈ節のベーシックブロックに、ステップ１１０２で生成したgoto命令またはthrow命令しか存在しなければTry-Catchブロックを取り外し、ステップ１１０１乃至ステップ１１０４による変形前の状態に戻す（ステップ１１０５）。

【００５５】

図１２は例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフ、図１３は図１２のコントロールフローグラフに対して図１１に示した動作により補償コードであるgoto命令またはThrow命令を挿入した状態を示すコントロールフローグラフを、それぞれ示す図である。

【００５６】

次に、本実施の形態におけるメモリへの書き込み命令の最適化の具体的な例を示す。

図１４は、Ｊａｖａのサンプルプログラムを示す図である。また、図１５は、図１４のサンプルプログラムを細かい処理単位に分けたプログラムである。

以下の説明において、mem_で始まる変数はメモリ上にある変数を指し、reg_で始まる変数はレジスタ上にある変数を指すものとする。また、図１５において、NULLCHECK reg_aとは、変数reg_aがnullかどうかをテストし、nullならば、例外処理であるNullPointerエクセプションを起こすような処理を表す。また、SIZECHECK reg_pos, reg_a[]とは、0 <= reg_pos < arraylength(reg_a)かどうかをテストし、条件に当てはまらなければ、例外処理であるArrayIndexOutOfBoundエクセプションを起こすような処理を表している。

【００５７】

図１６は、図１５に示すプログラムに対して、従来技術による最適化処理を行った状態を示す図である。

図１６において、斜体の文字で記述した部分１６０１、１６０２は、ループ内で実行される中間コードを表している。例外処理が起きる可能性がある中間コードがループ内に存在しているため、①の中間コードをループの外に出すことができない。

【００５８】

図１７は、図１６の状態のプログラムに対して、本実施の形態による補償コードの挿入を行った状態を示す図である。

図１７において、ループ内の例外処理が起きる可能性がある中間コードに対して、Try-Catchブロックを作成し、Ｃａｔｃｈ節内で受け取ったエクセプションをthrowしている（１７０１、１７０２参照）。

【００５９】

図１８は、図１７の状態のプログラムに対して、図１１に示した部分的な冗長度除去（Partial Redundancy Elimination）のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

図１８において、斜体の文字で記述した部分１８０１、１８０２、１８０３は、例外処理が起きないときにループ内で実行される中間コードを表している。例外処理が起きない場合のパスを考えると、図１６と比べてメモリへの書き込みが

なくなっている。

例外処理が起きた場合は、①にある中間コード（補償コード）によりメモリへの書き込みを行い、ループの外では②にある中間コード（補償コード）でメモリへの書き込みを行うことにより、正しい動作をする。さらに、例外処理が起きない場合においては、メモリへの書き込み命令がループ内に存在しないため、図１６と比べて高速に実行できる。

図１８の例では、Ｃａｔｃｈ節のベーシックブロックにthrow以外の中間コードが入ったため、図１１のステップ１１０５におけるTry-Catchブロックの取り外しは行われない。しかしながら、仮にＣａｔｃｈ節のベーシックブロック内にgoto命令またはthrow命令だけの中間コードしか残らないならば、図１９に示すようにして、Try-Catchブロックを取り外す。

【００６０】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入するさらに他の実施例として、関数呼び出しに関するメモリ書き込み命令の最適化について説明する。

従来、関数呼び出しを跨いで二つのメモリ書き込み命令の最適化を行う際には次の三つの条件が保証されていることが必要であった。

１．関数の中で最適化対象のメモリへの書き込みがない。

２．関数の中で最適化対象のメモリへの読み込みがない。

３．関数の中で例外処理が起こらない。

特に３の条件については、関数内で例外処理が起きる可能性があると判断されると、全てのメモリについての最適化が関数でKILLとして扱っていたため止まっていた。そのため、ほとんどの関数呼び出しを超えて最適化を行うことはできなかった。そこで、本実施の形態における補償コードの挿入により関数呼び出しのコントロールフローグラフを変形すると、例外処理が発生する場合について当該例外処理に対応するための動作を行うこととなるため、上記３番目の「関数の中で例外処理が起こらない」という条件を考慮する必要が無くなる。したがって、上記１番目及び２番目の条件が満たされていれば最適化が行えることとなる。

また、１番目及び２番目の条件を満足しない場合であっても、全てのメモリに

付いての最適化が止まるわけではなく、保証できなかったメモリのみデータフローアナリシス上でKILLとして扱えば良い。したがって、最適化の性能向上が期待できる。

【００６１】

図２０は、図１４のサンプルプログラムにおける配列アクセス部分を関数に置き換えたプログラムを示す図である。また、図２１は、図２０において、関数func内でmem_posに対する読み書きがないと分かった場合に、本実施の形態により補償コードを挿入してメモリの書き込み命令の最適化を行った状態を示す図である。

図２１において、斜体の文字で書いた部分２１０１、２１０２、２１０３が、例外処理が起きない場合にループ内で実行される部分である。図２０と図２１とを比較すると、ループ内においてmem_posへの書き込みがなくなっていることがわかる。

【００６２】

〔第２の実施の形態〕

次に、本発明における他の実施の形態について説明する。

本発明の第２の実施の形態は、図２に示した第１の実施の形態における最適化処理部１０と同様に構成され、前処理部１１、最適化処理実行部１２、後処理部１３を備える。

【００６３】

本実施の形態において、前処理部１１は、プログラム中の例外処理が起きる可能性のある命令を、ベーシックブロックとして、例外処理が起きるかどうかをチェックする命令と、実際に例外処理を起こす命令の二つに分ける。そして、これらの命令をコントロールフローグラフ上で明示的に分割する。そして、例外処理が起きる場合は、実際に例外処理を起こす命令に制御を移すようにする。

【００６４】

最適化処理実行部１２は、通常の最適化処理を実行する。最適化処理の対象であるプログラムは、前処理部１１により例外処理が起きるかどうかをチェックする命令と、実際に例外処理を起こす命令とに分けられている。このため、例外処

理を起こさない場合には、当該例外処理を考慮せず（当該例外処理を起こす命令のパスを通らずに）処理を行うこととなる。したがって、効果の高い最適化を実現することができる。

【００６５】

後処理部１３は、実際に例外処理を起こす命令があるベーシックブロックに、補償コードが生成されているかどうかをチェックする。そして、補償コードが生成されていない場合は、二つに分けた命令及びコントロールフローグラフを合成し、元の状態に戻す。

【００６６】

本実施の形態を実施するためには条件があるため、これについてまず説明する。例外処理が起きるかどうかをチェックするエクセプションチェック中間コードを実装する際には大きく分けて次の２つの実装方法がある。

第１は、ハードウェアの助けを借りて実装する方法である。第２は、明示的に例外処理をテストする命令を出し、条件分岐で例外処理が実際に起きる場合と起きない場合とで制御を分ける方法である。

【００６７】

第１のハードウェアを用いて実装する場合、配列境界チェックの実現を行っているならば、本実施の形態を適用すると、例外処理が起きない場合は、かえって処理が遅くなってしまう。したがって、本実施の形態では、第２の明示的に例外処理をテストする命令を出して、エクセプションチェック中間コードを実装することとする。

【００６８】

図２２は、本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

図２２を参照すると、オブジェクトコード（中間コード）に変換されたプログラムを入力した最適化処理部１０において、まず、前処理部１１が、メソッド内の例外処理が起きる可能性のある中間コードを例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける（ステップ２２０１）。そして、これらの中間コードを、コントロールフローグラフ上で

明示的に分割する（ステップ２２０２）。さらに、例外処理が起きる場合は、実際に例外処理を起こす中間コードに制御を移すようにコントロールフローグラフを構成する（ステップ２２０３）。

次に、最適化処理実行部１２が、最適化処理を実行する（ステップ２２０４）。

次に、後処理部１３が、前処理部１１において分割された実際に例外処理を起こす中間コードがあるベーシックブロックに補償コードが生成されているかどうかをチェックする（ステップ２２０５）。そして、生成されていない場合は、前処理部１１において分割された中間コード及びコントロールフローグラフを合成し、分割前の状態に戻す（ステップ２２０６）。

【００６９】

本実施の形態によれば、プログラムの実行時に例外処理が起きない場合、例外処理が起きる場合共に、本実施の形態を適用しない場合と比べて同等以上の実行速度が得られる。ただし、適用できる例外処理を起こす命令（エクセプションチェック中間コード）には制限がある。

そこで、本実施の形態は、上述した第１の実施の形態と組み合わせて用いることができる。具体的には、例えば次のような手順で、第１、第２の実施の形態を組み合わせて適用することができる。

１．プログラム中の変形対象であるエクセプションチェック中間コードが、オブジェクトコードレベルで例外処理が起きるかどうかをチェックする部分とエクセプションを実際に起こす部分とで構成されている場合は、本実施の形態を使って変形する。

２．その他のエクセプションチェック中間コードに対しては、第１の実施の形態（Try-Catchブロックを用いる方法）を使って変形する。

３．通常の最適化を行う。

４．最適化の対象とならなかった変形部分を元の中間コードに戻す。

【００７０】

次に、本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例として、第１の実施の形態においても説明した、部分的な冗長度除去（Pa

rtial Redundancy Elimination）のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法について説明する。

【００７１】

図２３は、本実施例における動作を説明するフローチャートである。

図２３を参照すると、まず、例外処理が起きる可能性のある中間コードを、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける（ステップ２３０１）。

次に、例外処理が起きるかどうかをチェックする中間コードの下でベーシックブロックを分割し、実際に例外処理を起こす中間コードがあるベーシックブロックと、例外処理が起きる可能性のある中間コードに後続する中間コードの二つのコードに、コントロールフローグラフのエッジを張る（ステップ２３０２）。例外処理が起きる場合は、実際に例外処理を起こす中間コードに制御を移す。

【００７２】

次に、ステップ２３０２で作成されたコントロールフローグラフに対して、メモリへの書き込み命令の最適化を行う（ステップ２３０３）。

最後に、ステップ２３０１で作成した実際に例外処理を起こす中間コードがあるベーシックブロックに、例外処理を起こす中間コードしか存在しないならば、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとを合成し、ステップ２３０１における分割前の状態に戻す（ステップ２３０４）。

【００７３】

図２４は例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフ、図２５は図２４のコントロールフローグラフに対して図２３に示した動作により補償コードを挿入し分岐を形成した状態のコントロールフローグラフを、それぞれ示す図である。

【００７４】

次に、本実施の形態におけるメモリへの書き込み命令の最適化の具体的な例を示す。

図２６は、図１４のサンプルプログラムに対して従来の最適化処理を実施して

得られた図１６に示す状態のプログラムを対象として、本実施の形態による補償コードの挿入を行った状態を示す図である。

図２６において、ループ内における例外処理が起きる可能性のある中間コードを、例外処理が起きるかどうかをチェックする中間コードSIZECHECK_FAILと、実際に例外処理を起こす中間コードSIZECHECK_EXCEPTIONの二つに分けている（２６０１、２６０２参照）。SIZECHECK_FAILは、例外処理が起きる場合にTRUEを返す中間コードを表している。

【００７５】

図２７は、図２６の状態のプログラムに対して、図２３に示した部分的な冗長度除去（Partial Redundancy Elimination）のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

図２７において、斜体の文字で書いてある部分２７０１、２７０２、２７０３は、例外処理が起きないときにループ内で実行される中間コードを表している。エクセプションが起きない場合のパスを考えると、図１６と比べてメモリへの書き込みがなくなっている。

例外処理が起きた場合は、①にある中間コード（補償コード）によりメモリへの書き込みを行い、ループの外では②にある中間コード（補償コード）でメモリに書き込みを行うことにより、正しい動作をする。さらに、例外処理が起きない場合においては、メモリへの書き込み命令がループ内に存在しないため、図１６と比べて高速に実行できる。

図２７の例では、例外処理を起こす中間コードがあるベーシックブロックに例外処理を起こす中間コード以外の中間コードが入ったため、図２３のステップ２３０４における中間コードを合成し分割前の状態に戻す処理は行われない。しかしながら、仮に例外処理を起こす中間コードがあるベーシックブロックに例外処理を起こす中間コードしか残らないならば、図２８に示すようにして、分割前の状態に戻す。

【００７６】

【発明の効果】

以上説明したように、本発明によれば、例外処理が起きる可能性のある命令に

対し、例外発生時、補償コードを経た後に当該例外処理を実行するようにプログラムを変形することにより、効果の高い最適化を行うことができる。

【図面の簡単な説明】

【図１】　本発明により、例外処理が起きる可能性のある命令において、エクセプション・ハンドラへ移行する様子を示す図である。

【図２】　第１の実施の形態によるコンパイラにおける最適化処理部の構成を説明する図である。

【図３】　本実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

【図４】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、ベーシックブロックにおける要素の集合を求めるためのアルゴリズムを示す図である。

【図５】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、データフローアナリシスを行うための関係式を示す図である。

【図６】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、ローカル変数の集合を求めるためのアルゴリズムを示す図である。

【図７】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例を説明する図であり、所定の命令に対して新たなエクセプション・ハンドラを設定するためのアルゴリズムを示す図である。

【図８】　図７のアルゴリズムにおける＜この命令について新たなエクセプション・ハンドラを設定する。＞の処理内容を説明するフローチャートである。

【図９】　配列内の最小値及び最大値を求めるＪａｖａのサンプルプログラムを示す図である。

【図１０】　図９のプログラムに対して本実施の形態により補償コードを挿入する変形を加えた状態を示す図である。

【図１１】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する他の実施例として、部分的な冗長度除去のアルゴリズムを使ったメモリ

への書き込み命令の最適化の効果を高める方法を説明するフローチャートである。

【図１２】　例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフを示す図である。

【図１３】　図１２のコントロールフローグラフに対して補償コードを挿入した状態を示すコントロールフローグラフを示す図である。

【図１４】　Ｊａｖａのサンプルプログラムを示す図である。

【図１５】　図１４のサンプルプログラムを細かい処理単位に分けたプログラムを示す図である。

【図１６】　図１５に示すプログラムに対して、従来技術による最適化処理を行った状態を示す図である。

【図１７】　図１６の状態のプログラムに対して、本実施の形態による補償コードの挿入を行った状態を示す図である。

【図１８】　図１７の状態のプログラムに対し、部分的な冗長度除去のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

【図１９】　図１８の状態のプログラムからTry-Catchブロックを取り外す様子を示す図である。

【図２０】　図１４のサンプルプログラムにおける配列アクセス部分を関数に置き換えたプログラムを示す図である。

【図２１】　図２０において、補償コードを挿入してメモリの書き込み命令の最適化を行った状態を示す図である。

【図２２】　第２の実施の形態における最適化処理の概略的な流れを説明するフローチャートである。

【図２３】　本実施の形態を用いて例外処理に対応するための補償コードを挿入する実施例として、部分的な冗長度除去のアルゴリズムを使ったメモリへの書き込み命令の最適化の効果を高める方法を説明するフローチャートである。

【図２４】　例外処理が起きる可能性のある中間コードを含むプログラムのコントロールフローグラフを示す図である。

【図２５】　図２４のコントロールフローグラフに対して補償コードを挿入

し分岐を形成した状態のコントロールフローグラフを示す図である。

【図２６】　図１６に示す状態のプログラムを対象として、本実施の形態による補償コードの挿入を行った状態を示す図である。

【図２７】　図２６の状態のプログラムに対して、部分的な冗長度除去のアルゴリズムを用いてメモリ書き込み命令の最適化を行った状態を示す図である。

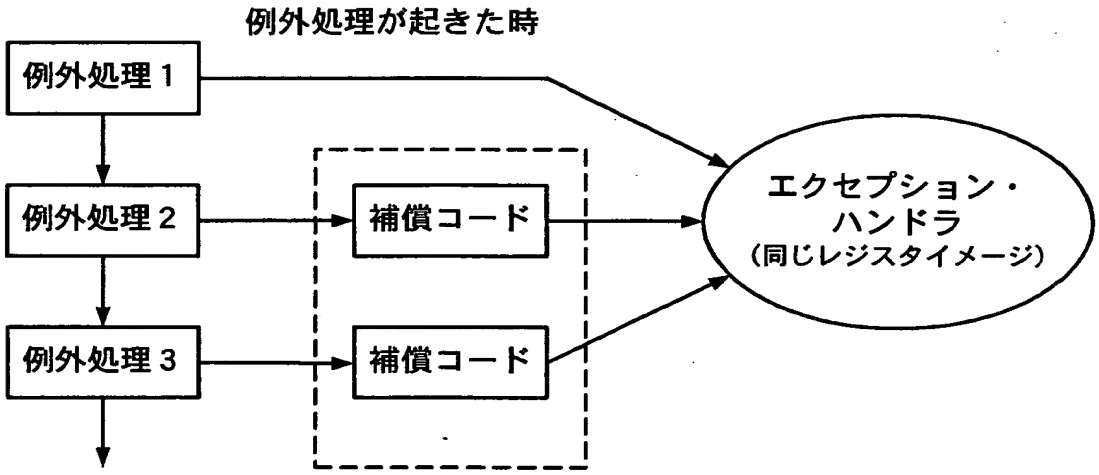
【図２８】　図２７の状態のプログラムから不要な分岐を分割前の状態に戻す様子を示す図である。

【符号の説明】

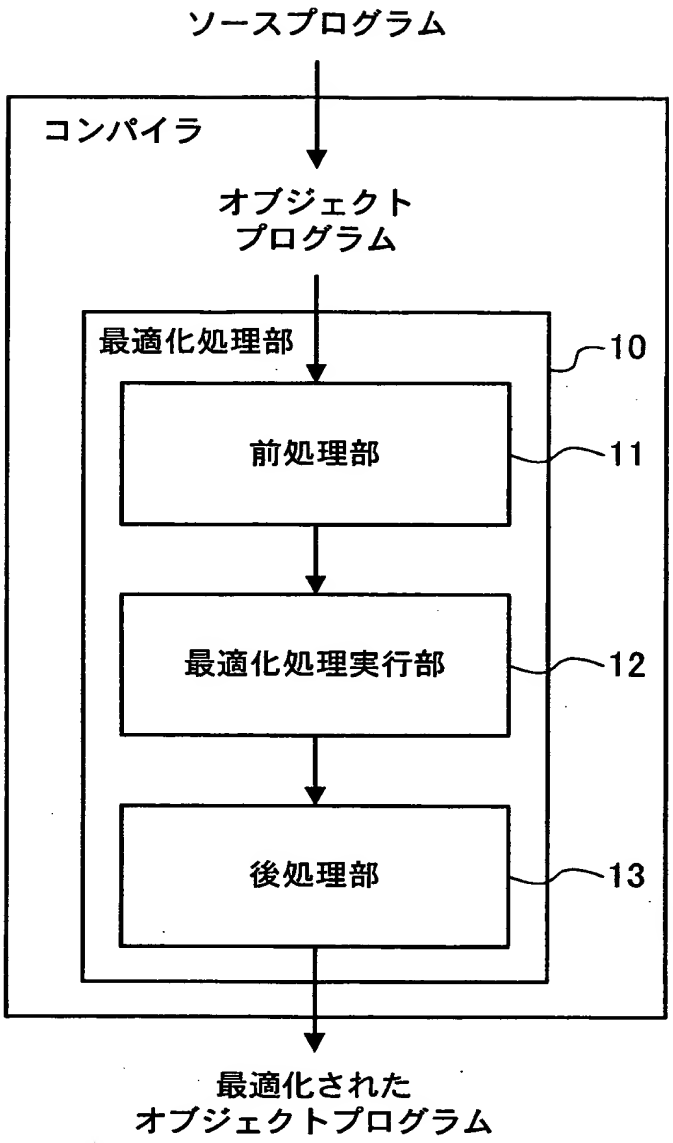
１０…最適化処理部、１１…前処理部、１２…最適化処理実行部、１３…後処理部

【書類名】　　図面

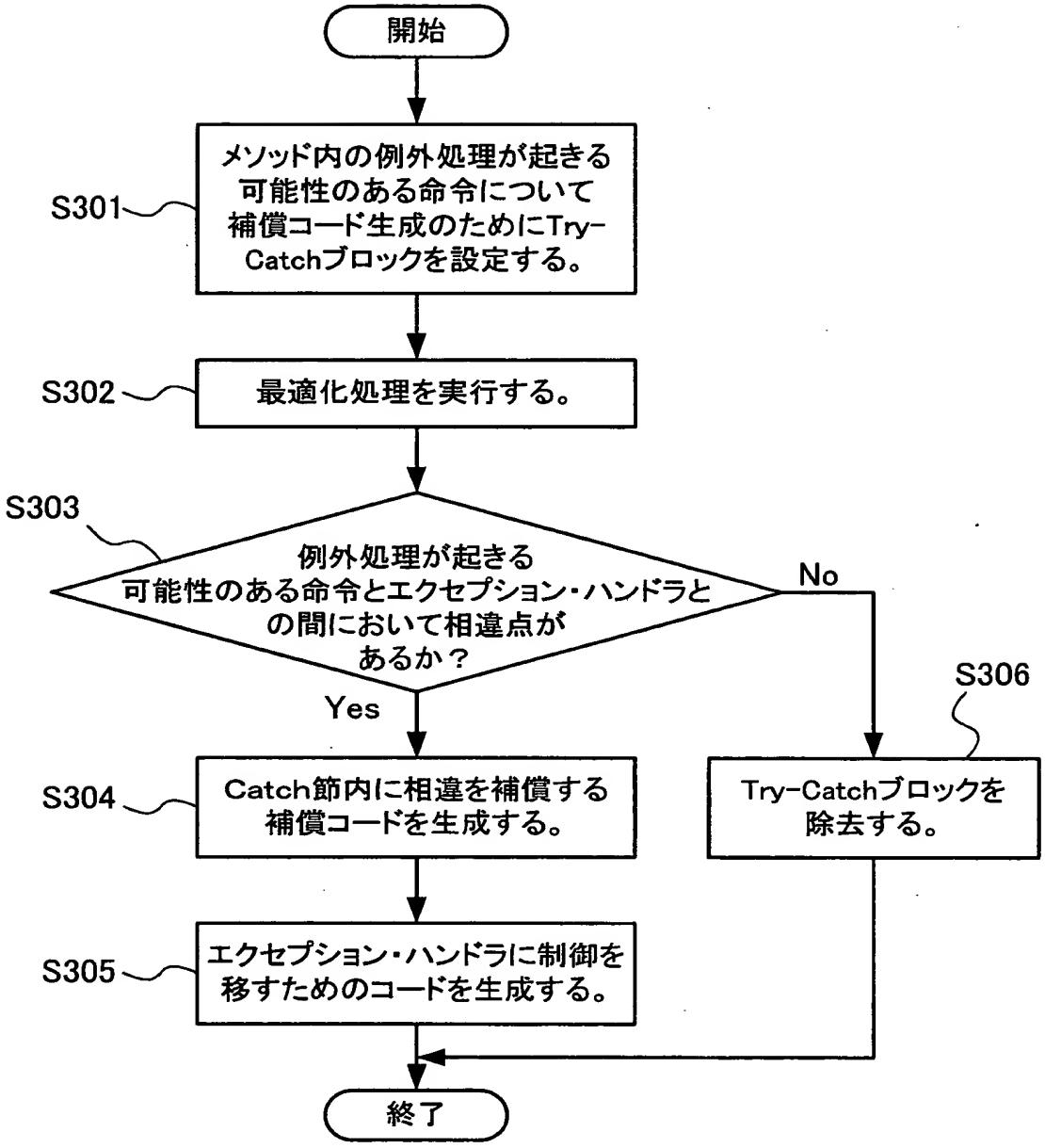
【図１】


例外処理が起きた時
例外処理１
例外処理２
例外処理３
補償コード
補償コード
エクセプション・
ハンドラ
(同じレジスタイメージ)

【図２】


ソースプログラム
コンパイラ
オブジェクト
プログラム
最適化処理部
10
前処理部
11
最適化処理実行部
12
後処理部
13
最適化された
オブジェクトプログラム

【図３】


開始
メソッド内の例外処理が起きる可能性のある命令について補償コード生成のためにTry-Catchブロックを設定する。
S301
最適化処理を実行する。
S302
S303
例外処理が起きる可能性のある命令とエクセプション・ハンドラとの間において相違点があるか？
No
Yes
S306
Catch節内に相違を補償する補償コードを生成する。
S304
Try-Catchブロックを除去する。
エクセプション・ハンドラに制御を移すためのコードを生成する。
S305
終了

【図４】

```
for (すべてのbasic blockについて繰り返す) {
    bbidx = basic blockのインデックス
    if (このbasic blockがエクセプション・ハンドラの開始点) {
        reach[bbidx].gen = ( このエクセプション・ハンドラに対応する全てのTRY,
            すべてのローカル変数 );
    } else {
        reach[bbidx].gen = φ
    }
    reach[bbidx].kill = φ
    for (basic block内のすべての命令について実行順に繰り返す) {
        if (命令 == ローカル変数へのwrite) {
            reach[bbidx].kill U= ( 全てのTRY, 書き込まれたローカル変数番号 );
        }
    }
}
```

【図５】

$$reach[B].in = \bigcup_{P \in Pred(B)} reach[P].out \cup reach[B].gen$$

$$reach[B].out = reach[B].in - reach[B].kill$$

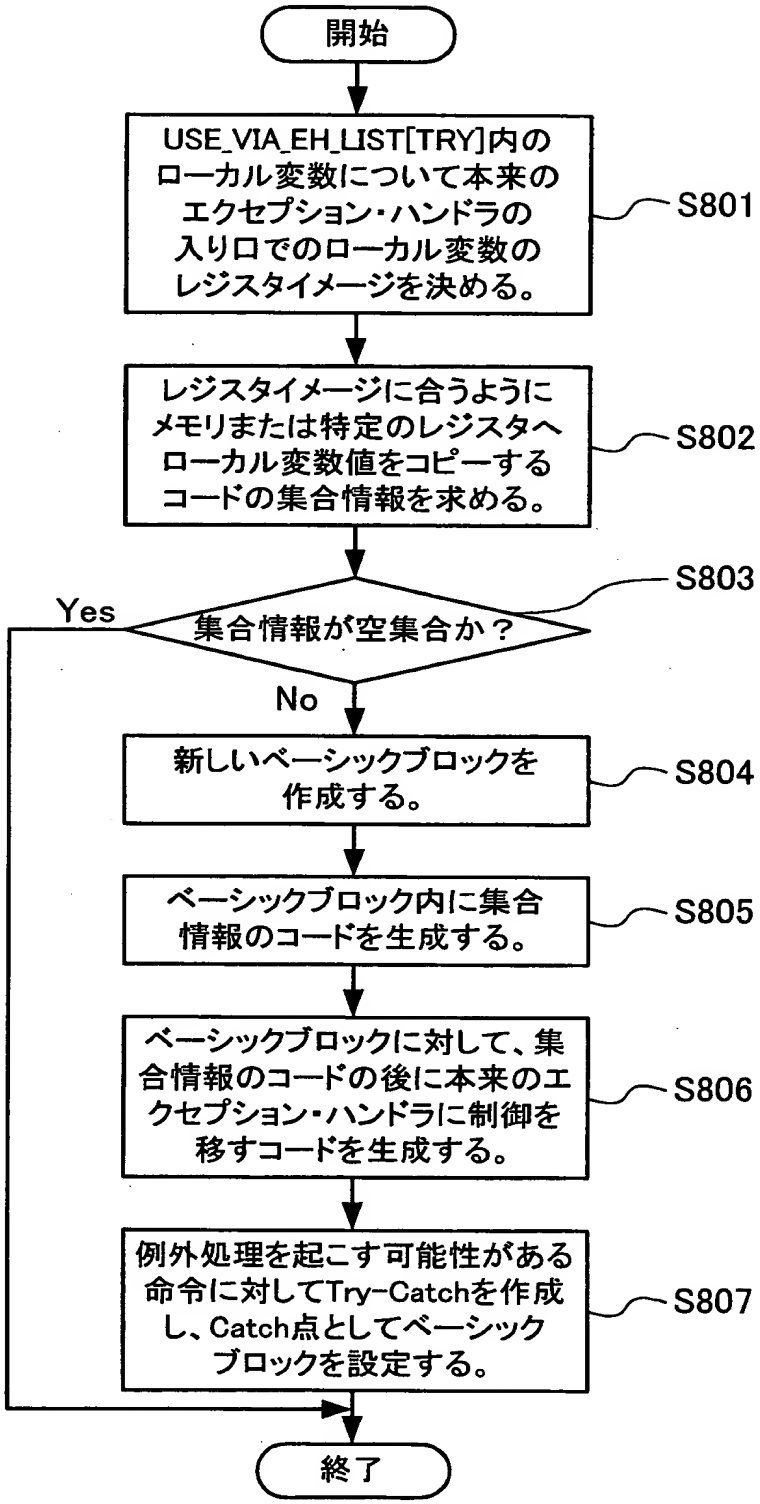
【図６】

```
for (全てのTRYについて繰り返す) USE_VIA_EH_LIST[TRY] = φ;
for (すべてのbasic blockについて繰り返す) {
    bbidx = basic blockのインデックス;
    if (reach[bbidx].in != φ) {
        for (basic block内のすべての命令について実行順に繰り返す) {
            switch(命令) {
            case ローカル変数へのwrite:
                reach[bbidx].in -= ( 全てのTRY, 書き込まれたローカル変数番号 );
                break;
            case ローカル変数からのread:
                for (全てのTRYについて繰り返す) {
                    if ( (TRY, 読み込まれたローカル変数番号 ) ∈ reach[bbidx].in) {
                        USE_VIA_EH_LIST[TRY] ∪= 読み込まれたローカル変数番号;
                    }
                }
                break;
            }
        }
    }
}
```

【図７】

```
for (すべてのbasic blockについて繰り返す) {
    if (basic blockがtry regionに含まれる) {
        TRY = Try Region の識別番号
        if (USE_VIA_EH_LIST[TRY] != φ) {
            bbidx = basic blockのインデックス;
            for (basic block内のすべての命令について実行順に繰り返す) {
                if (命令 ∈ エクセプションを起こす可能性がある命令) {
                    if (起きる可能性があるｴｸｾﾌﾟｼｮﾝに対応するｴｸｾﾌﾟｼｮﾝ･ﾊﾝﾄﾞﾗがある) {
                        < この命令について新たなｴｸｾﾌﾟｼｮﾝ･ﾊﾝﾄﾞﾗを設定する。 >
                    }
                }
            }
        }
    }
}
```

【図８】


開始
USE_VIA_EH_LIST[TRY]内のローカル変数について本来のエクセプション・ハンドラの入り口でのローカル変数のレジスタイメージを決める。
S801
レジスタイメージに合うようにメモリまたは特定のレジスタへローカル変数値をコピーするコードの集合情報を求める。
S802
S803
集合情報が空集合か？
Yes
No
新しいベーシックブロックを作成する。
S804
ベーシックブロック内に集合情報のコードを生成する。
S805
ベーシックブロックに対して、集合情報のコードの後に本来のエクセプション・ハンドラに制御を移すコードを生成する。
S806
例外処理を起こす可能性がある命令に対してTry-Catchを作成し、Catch点としてベーシックブロックを設定する。
S807
終了

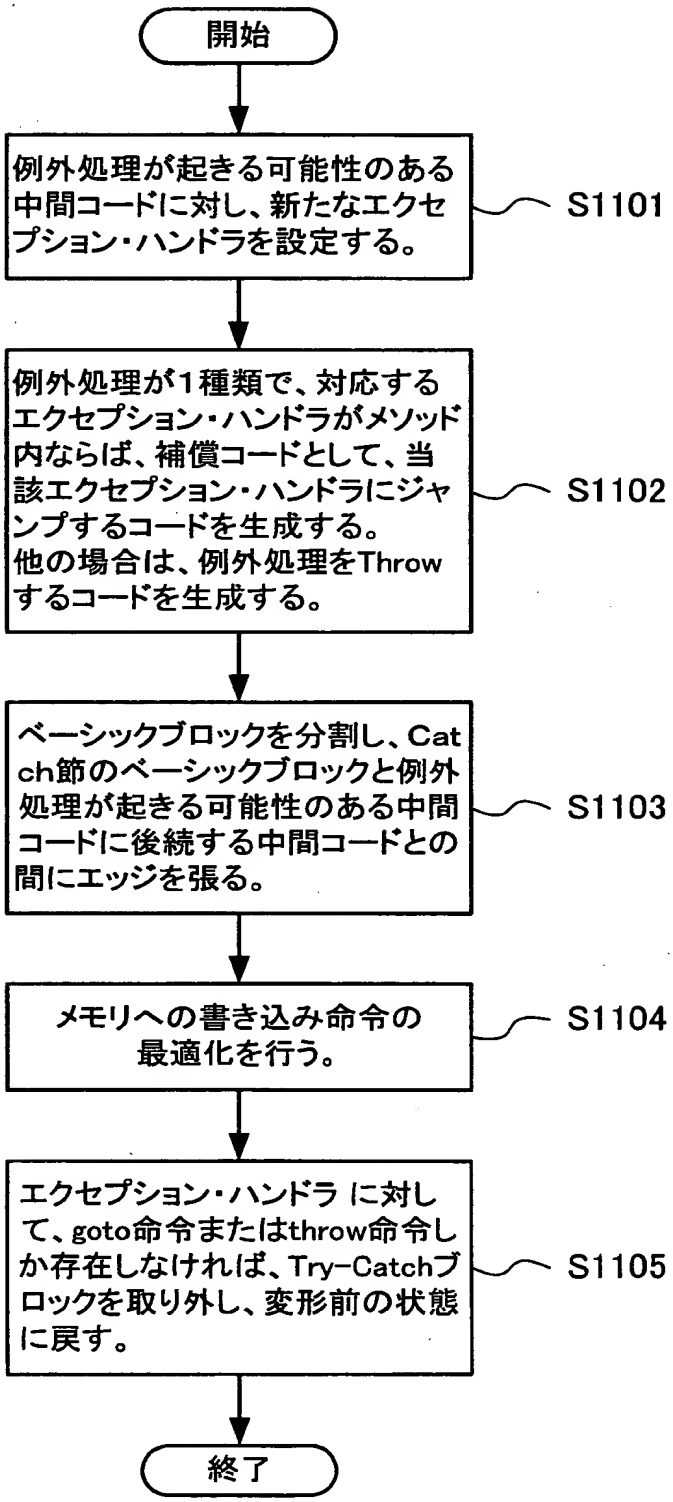
【図９】

```
int MIN_VAL, MAX_VAL;
void sample(int a[][], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          --- (1)
    j = 0;          --- (2)
    try {
        min = a[i][j];    --- (3)
        max = min;        --- (4)
        i = 0;            --- (5)
        while(i < size_x) {        --- (6)
            j = 0;                 --- (7)
            while(j < size_y) {    --- (8)
                int val = a[i][j];         --- (9)
                if (min > val) {   --- (10)
                    min = val;     --- (11)
                }
                if (max < val) {   --- (12)
                    max = val;     --- (13)
                }
                j++;               --- (14)
            }
            i++;           --- (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("ArrayIndexOutOfBounds i=" + i + " j=" + j);     --- (16)
        /* error status : min > max */
        max = 0x80000000;--- (17)
        min = 0x7FFFFFFF;--- (18)
    }
    MIN_VAL = min;                 --- (19)
    MAX_VAL = max;                 --- (20)
}
```

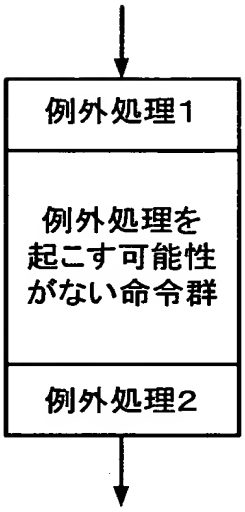
【図１０】

```
int MIN_VAL, MAX_VAL;
void sample(int a[][], int size_x, int size_y) {
    int min, max;
    int i, j;
    i = 0;          ── (1)
    j = 0;          ── (2)
    try {
        NULLCHECK a;                    ── (3.1)
        try {
            SIZECHECK i, a[];           ── (3.2)
        } catch(ArrayIndexOutOfBoundsExcetion e) {
            i と j の変数値をそれぞれR1, R2にコピーする。
            goto Handler;
        }
        try {
            SIZECHECK j, a[i];          ── (3.3)
        } catch(ArrayIndexOutOfBoundsException e) {
            i と j の変数値をそれぞれR1, R2にコピーする。
            goto Handler;
        }
        min = a[i][j];          ── (3.4)
        max = min; ── (4)
        i = 0;                  ── (5)
        while(i < size_x) {     ── (6)
            j = 0;              ── (7)
            while(j < size_y) {         ── (8)
                try {
                    SIZECHECK i, a[]; ── (9.1)
                } catch(ArrayIndexOutOfBoundsException e) {
                    i と j の変数値をそれぞれR1, R2にコピーする。
                    goto Handler;
                }
                try {
                    SIZECHECK j, a[i];          ── (9.2)
                } catch(ArrayIndexOutOfBoundsException e) {
                    i と j の変数値をそれぞれR1, R2にコピーする。
                    goto Handler;
                }
                int val = a[i][j];      ── (9.3)
                if (min > val) {        ── (10)
                    min = val;          ── (11)
                }
                if (max < val) {        ── (12)
                    max = val;          ── (13)
                }
                j++;                    ── (14)
            }
            i++;                ── (15)
        }
    } catch (ArrayIndexOutOfBoundsException e) {
    Handler:
        /* i, jの値はそれぞれR1, R2に入っている。 */
        System.err.println("ArrayIndexOutOfBounds i=" + i + " j=" + j);    ── (16)
        /* error status : min > max */
        max = 0x80000000;   ── (17)
        min = 0x7FFFFFFF;   ── (18)
    }
    MIN_VAL = min;          ── (19)
    MAX_VAL = max;          ── (20)
}
```

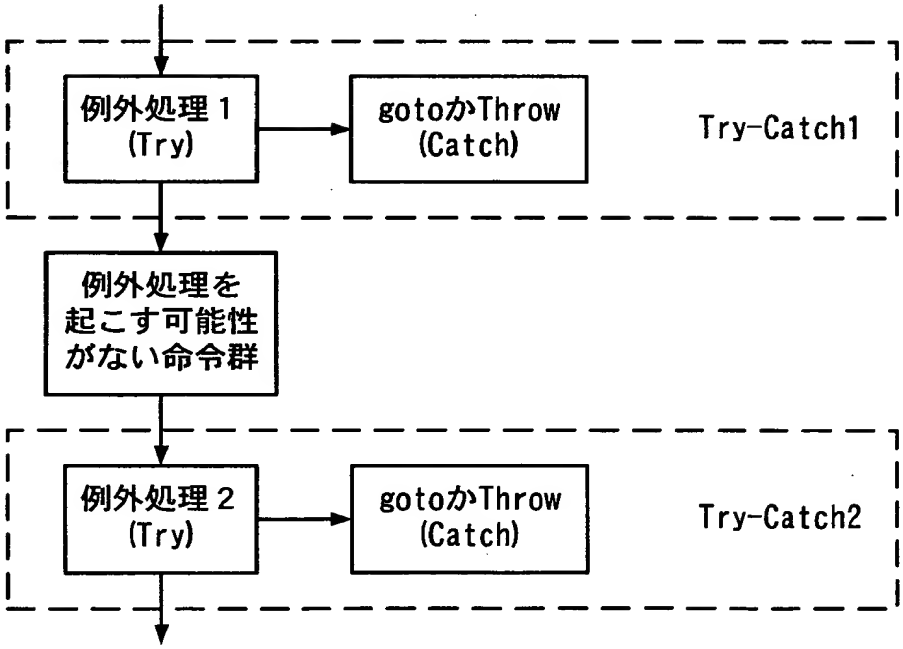
【図１１】


開始
例外処理が起きる可能性のある中間コードに対し、新たなエクセプション・ハンドラを設定する。
S1101
例外処理が1種類で、対応するエクセプション・ハンドラがメソッド内ならば、補償コードとして、当該エクセプション・ハンドラにジャンプするコードを生成する。
他の場合は、例外処理をThrowするコードを生成する。
S1102
ベーシックブロックを分割し、Catch節のベーシックブロックと例外処理が起きる可能性のある中間コードに後続する中間コードとの間にエッジを張る。
S1103
メモリへの書き込み命令の最適化を行う。
S1104
エクセプション・ハンドラ に対して、goto命令またはthrow命令しか存在しなければ、Try-Catchブロックを取り外し、変形前の状態に戻す。
S1105
終了

【図１２】


例外処理1
例外処理を
起こす可能性
がない命令群
例外処理2

【図１３】


例外処理１
(Try)
gotoかThrow
(Catch)
Try-Catch1
例外処理を
起こす可能性
がない命令群
例外処理２
(Try)
gotoかThrow
(Catch)
Try-Catch2

【図１４】

```
static int mem_pos;       // メモリ上にある変数
static int mem_a[ ];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while(mem_a[mem_pos] != reg_data) {
        mem_pos ++;
    }
}
```

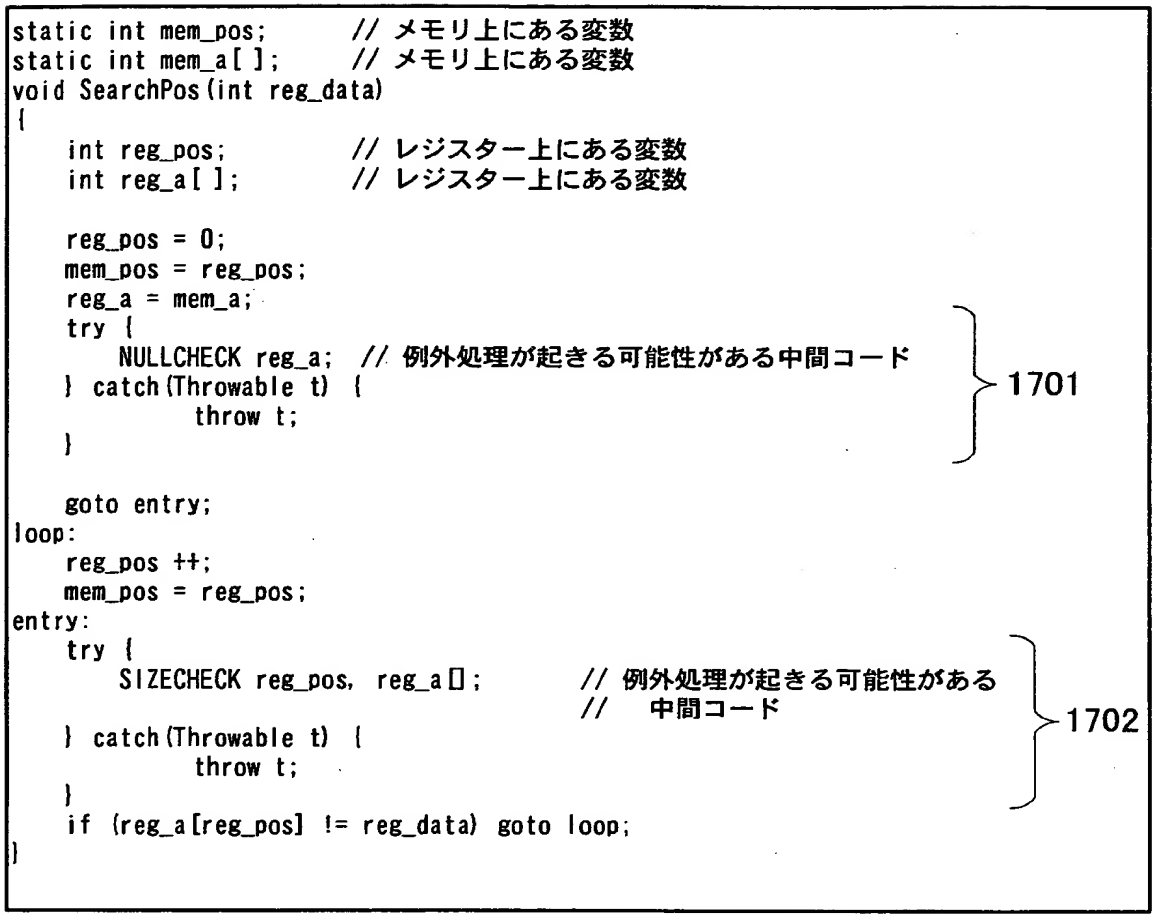
【図１５】

```
static int mem_pos;       // メモリ上にある変数
static int mem_a[ ];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[ ];         // レジスター上にある変数

    mem_pos = 0;

    goto entry;
loop:
    reg_pos = mem_pos;
    reg_pos ++;
    mem_pos = reg_pos;
entry:
    reg_pos = mem_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;              // 例外処理が起きる可能性がある中間コード
    SIZECHECK reg_pos, reg_a[];   // 例外処理が起きる可能性がある中間コード
    if (reg_a[reg_pos] != reg_data) goto loop;
}
```

【図１６】

```
static int mem_pos;        // メモリ上にある変数
static int mem_a[ ];       // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;           // レジスター上にある変数
    int reg_a[ ];          // レジスター上にある変数

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    NULLCHECK reg_a;              // 例外処理が起きる可能性がある中間コード

    goto entry;                                    1601
loop:
    reg_pos ++;
    mem_pos = reg_pos; ―① // この中間コードをループ内から出すことができない
entry:
    SIZECHECK reg_pos, reg_a[];   // 例外処理が起きる可能性がある中間コード
    if (reg_a[reg_pos] != reg_data) goto loop;
}
                                                   1602
```

【図１７】


static int mem_pos;　　// メモリ上にある変数
static int mem_a[ ];　　// メモリ上にある変数
void SearchPos(int reg_data)
{
int reg_pos;　　// レジスター上にある変数
int reg_a[ ];　　// レジスター上にある変数
reg_pos = 0;
mem_pos = reg_pos;
reg_a = mem_a;
try {
NULLCHECK reg_a;　// 例外処理が起きる可能性がある中間コード
} catch(Throwable t) {
throw t;
}
1701
goto entry;
loop:
reg_pos ++;
mem_pos = reg_pos;
entry:
try {
SIZECHECK reg_pos, reg_a[];　　// 例外処理が起きる可能性がある
//　中間コード
} catch(Throwable t) {
throw t;
}
1702
if (reg_a[reg_pos] != reg_data) goto loop;
}

【図１８】

```
static int mem_pos;        // メモリ上にある変数
static int mem_a[ ];       // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;           // レジスター上にある変数
    int reg_a[ ];          // レジスター上にある変数

    reg_pos = 0;
    reg_a = mem_a;
    try {
        NULLCHECK reg_a; // 例外処理が起きる可能性がある中間コード
    } catch(Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }

    goto entry;                    1801
loop:
    reg_pos ++;                                  1802
entry:
    try {
        SIZECHECK reg_pos, reg_a[];      // 例外処理が起きる可能性がある
                                         // 中間コード
    } catch(Throwable t) {
        mem_pos = reg_pos;          —①
        throw t;
    }
    if (reg_a[reg_pos] != reg_data) goto loop;      1803
    mem_pos = reg_pos;     —②
}
```

【図１９】

```
try {
    ExceptionCheck;        // 例外処理が起きる可能性がある中間コード
} catch(Throwable t) {
    throw t;
}
```

↓

```
ExceptionCheck;  // 例外処理が起きる可能性がある中間コード
```

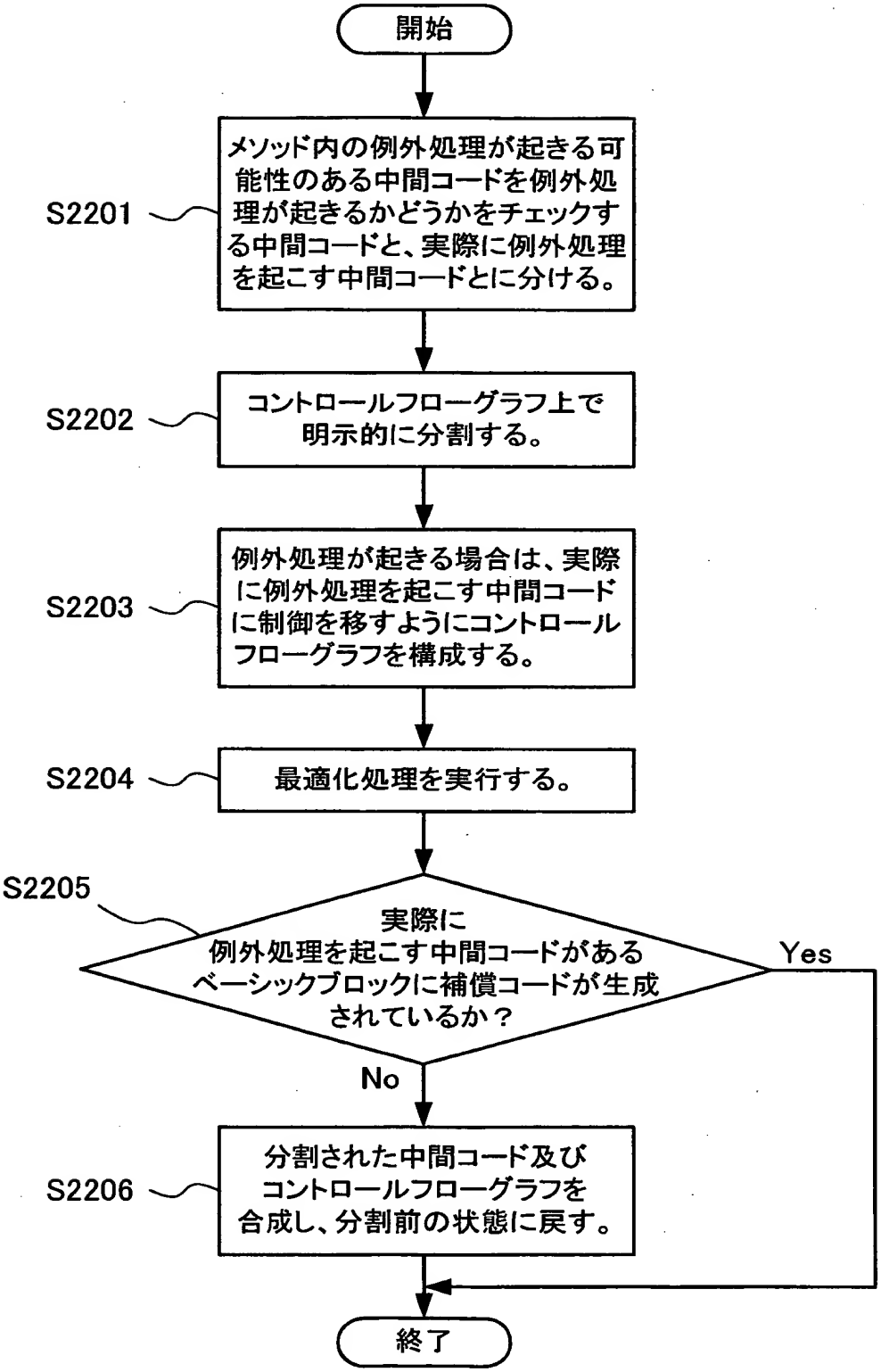
【図２０】

```
static int mem_pos;       // メモリ上にある変数
void SearchPos(int reg_data)
{
    mem_pos = 0;
    while(func(mem_pos)  != reg_data) {
        mem_pos ++;
    }
}
```

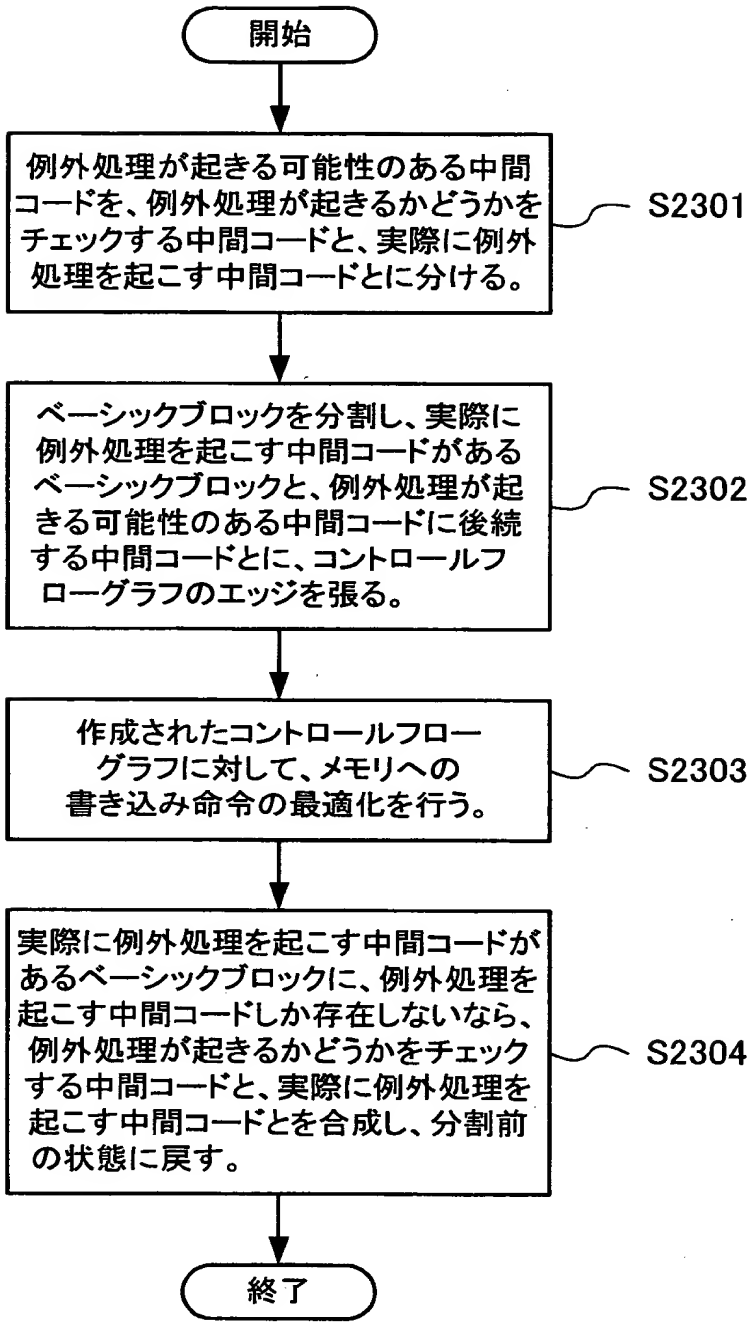
【図２１】

```
static int mem_pos;       // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos; // レジスター上にある変数
    int reg_ret; // レジスター上にある変数

    reg_pos = 0;

    goto entry;
loop:
    reg_pos ++;          ← 2101
entry:
    try {
        reg_ret = func(reg_pos);          ← 2102
    } catch(Throwable t) {
        mem_pos = reg_pos;
        throw t;
    }
    if (reg_ret != reg_data) goto loop;          ← 2103
    mem_pos = reg_pos;
}
```

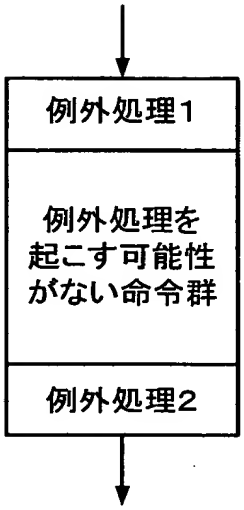
【図２２】


開始
メソッド内の例外処理が起きる可能性のある中間コードを例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける。
S2201
コントロールフローグラフ上で明示的に分割する。
S2202
例外処理が起きる場合は、実際に例外処理を起こす中間コードに制御を移すようにコントロールフローグラフを構成する。
S2203
最適化処理を実行する。
S2204
S2205
実際に例外処理を起こす中間コードがあるベーシックブロックに補償コードが生成されているか？
Yes
No
分割された中間コード及びコントロールフローグラフを合成し、分割前の状態に戻す。
S2206
終了

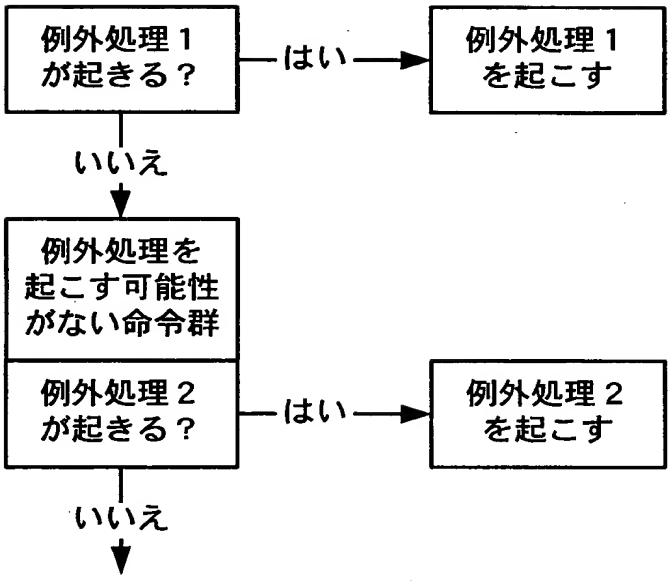
【図２３】


開始
例外処理が起きる可能性のある中間コードを、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとに分ける。
S2301
ベーシックブロックを分割し、実際に例外処理を起こす中間コードがあるベーシックブロックと、例外処理が起きる可能性のある中間コードに後続する中間コードとに、コントロールフローグラフのエッジを張る。
S2302
作成されたコントロールフローグラフに対して、メモリへの書き込み命令の最適化を行う。
S2303
実際に例外処理を起こす中間コードがあるベーシックブロックに、例外処理を起こす中間コードしか存在しないなら、例外処理が起きるかどうかをチェックする中間コードと、実際に例外処理を起こす中間コードとを合成し、分割前の状態に戻す。
S2304
終了

【図２４】


例外処理1
例外処理を
起こす可能性
がない命令群
例外処理2

【図２５】


例外処理１
が起きる？
はい
例外処理１
を起こす
いいえ
例外処理を
起こす可能性
がない命令群
例外処理２
が起きる？
はい
例外処理２
を起こす
いいえ

【図２６】

```
static int mem_pos;       // メモリ上にある変数
static int mem_a[ ];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[ ];         // レジスター上にある変数

    reg_pos = 0;
    mem_pos = reg_pos;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) {    // 例外処理が起きるかどうか
                                   //    チェックする中間コード
        NULLCHECK_EXCEPTION reg_a;        // 例外処理を起こす中間コード
    }

    goto entry;
loop:
    reg_pos ++;                                    2601
    mem_pos = reg_pos;
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) {         // 例外処理が起きるかどうか
                                                   //    チェックする中間コード
        SIZECHECK_EXCEPTION reg_pos, reg_a[];      // 例外処理を起こす中間コード
    }
    if (reg_a[reg_pos] != reg_data) goto loop;          2602
}
```

【図２７】

```
static int mem_pos;       // メモリ上にある変数
static int mem_a[ ];      // メモリ上にある変数
void SearchPos(int reg_data)
{
    int reg_pos;          // レジスター上にある変数
    int reg_a[ ];         // レジスター上にある変数

    reg_pos = 0;
    reg_a = mem_a;
    if (NULLCHECK_FAIL reg_a) {    // 例外処理が起きるかどうか
                                   //    チェックする中間コード
        mem_pos = reg_pos;
        NULLCHECK_EXCEPTION reg_a;         // 例外処理を起こす中間コード
    }

    goto entry;
loop:                                                    2702
    reg_pos ++;      <~~ 2701                           <~~
entry:
    if (SIZECHECK_FAIL reg_pos, reg_a[]) {          // 例外処理が起きるかどうか
                                                    //   チェックする中間コード
        mem_pos = reg_pos;          -①
        SIZECHECK_EXCEPTION reg_pos, reg_a[];       // 例外処理を起こす中間コード
    }
    if (reg_a[reg_pos] != reg_data) goto loop;
    mem_pos = reg_pos;    -②                    <~~ 2703
}
```

【図２８】

```
if (EXCEPTIONCHECK_FAIL) {          // 例外処理が起きるかどうか
                                    //    チェックする中間コード
    EXCEPTION;                      // 例外処理を起こす中間コード
}
```

↓

```
ExceptionCheck;  // 例外処理が起きる可能性がある中間コード
```

【書類名】　　要約書

【要約】

【課題】　例外処理が起きる可能性のある命令を含むプログラムに対して効果の高い最適化を行う。

【解決手段】　プログラミング言語で記述されたプログラムのソース・コードを機械語に変換するコンパイラにおいて、機械語に変換されたオブジェクトプログラムに対して最適化処理を行う最適化処理実行部１２と、このオブジェクトプログラムに対して、このオブジェクトプログラム中の例外処理が起きる可能性のある命令に関する例外処理の発生点とこの例外処理を実行する場所とにおける内容的な相違を吸収するための変形を行う前処理部１１及び後処理部１３とを備える。

【選択図】　　図２

# 認定・付加情報

| | |
|---|---|
| 特許出願の番号 | 特願２０００－１１４１９３ |
| 受付番号 | ５００００４７７２５１ |
| 書類名 | 特許願 |
| 担当官 | 佐藤　一博　　１９０９ |
| 作成日 | 平成１２年　５月３１日 |

＜認定情報・付加情報＞

【特許出願人】

【識別番号】　390009531

【住所又は居所】　アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）

【氏名又は名称】　インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】　100086243

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　大和事業所内

【氏名又は名称】　坂口　博

【復代理人】　申請人

【識別番号】　100104880

【住所又は居所】　東京都港区赤坂７－１０－９　第４文成ビル２０２セリオ国際特許事務所

【氏名又は名称】　古部　次郎

【選任した代理人】

【識別番号】　100091568

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　大和事業所内

【氏名又は名称】　市位　嘉宏

【選任した復代理人】

【識別番号】　100100077

【住所又は居所】　東京都港区赤坂７－１０－９　第４文成ビル２０２セリオ国際特許事務所

【氏名又は名称】　大場　充

次頁無

# 出　願　人　履　歴　情　報

識別番号　　〔３９０００９５３１〕

１．変更年月日　　１９９０年１０月２４日
［変更理由］　新規登録
住　所　　アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）
氏　名　　インターナショナル・ビジネス・マシーンズ・コーポレイション

２．変更年月日　　２０００年　５月１６日
［変更理由］　名称変更
住　所　　アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）
氏　名　　インターナショナル・ビジネス・マシーンズ・コーポレーション